Politecnico di Torino

Corso di Laurea in Matematica per l'Ingegneria

Discrete Fracture Network

Progetto del corso di Programmazione e Calcolo Scientifico 2024

Relatori:

Stefano Berrone
Matteo Cicuttin
Gioana Teora
Fabio Vicini

Candidati:

Marco Buffo Blin - 297583
Giulia Calabrese - 294808
Sofia Silvestro - 281693

Sommario

1. INTRODUZIONE AL PROBLEMA

Un Discrete Fracture Network (DFN) è un metodo utilizzato in geologia per studiare i modelli di flusso attraverso le fratture presenti nelle rocce al cui interno possono scorrere dei fluidi.
Nell'analisi di una popolazione di fratture si devono individuare le fratture isolate, ovvero quelle che non si intersecano con altre fratture, in modo da escluderle a priori nella valutazione delle intersezioni. In questo contesto si è tralasciata l'analisi delle fratture che presentano come intersezione un segmento di dimensione nulla, vale a dire un singolo punto.

Un DFN è costituito da un insieme di fratture, in questo particolare caso rappresentate da una serie di poligoni planari che potrebbero intersecarsi nello spazio tridimensionale. Queste intersezioni sono chiamate tracce, corrispondenti a segmenti di diversa lunghezza, eventualmente nulla o ridotta ad un unico punto.
Le tracce si possono classificare come segue:

1. Traccia passante: un segmento con entrambi gli estremi giacenti sul bordo della frattura
2. Traccia non passante: segmento che presenta almeno un estremo appartenente alla superficie interna della frattura stessa.

Questo tipo di classificazione per le tracce è riferito ad ogni poligono: una traccia può risultare passante per uno dei due poligoni che la genera e non passante per il secondo poligono che contribuisce alla sua creazione.

In questa prima parte del progetto proposto si ha l'obiettivo di identificare e classificare le tracce presenti all'interno di ciascuna frattura. Inizialmente si legge un file contenente una serie di poligoni identificati da un numero intero e descritti dai vertici che li compongono nello spazio.
Si è poi utilizzato il concetto di baricentro e di sfera circoscritta per valutare la posizione reciproca tra poligoni.
Successivamente si utilizzano delle funzioni specifiche per il calcolo dell'equazione del piano su cui giace un poligono, dell'equazione di una retta generata dall'intersezione tra due piani e dell'equazione di una retta passante per due punti, corrispondenti a due dei vertici di un poligono.
Un'ulteriore funzione è stata implementata per calcolare l'ascissa curvilinea delle rette ricavate in precedenza, sfruttando il metodo di risoluzione dei sistemi lineari con decomposizione QR.
Dopo aver popolato le strutture dati relative a frattura e traccia, si salva tutto nel file *'Traces.txt'*. Ogni file derivante dall'esecuzione relativa ai diversi file di input forniti dal progetto, viene poi riportato all'interno di una cartella che raccoglie i risultati finali.

Infine, per la validazione delle singole unità logiche vengono utilizzati i test della libreria esterna dei GoogleTest per verificare la correttezza delle singole funzioni implementate.

La seconda richiesta del progetto è quella di individuare l'insieme dei sotto-poligoni che vengono ricavati su ogni frattura a seguito del suo taglio seguendo tutte le tracce presenti su di essa.
In precedenza per ogni poligono è stato salvato, con un ordine ben preciso, l'insieme di tracce presenti su di esso. Rispettando il medesimo ordinamento viene suddivisa ogni frattura: inizialmente si considerano le tracce passanti poi quelle non passanti, entrambe in ordine di lunghezza. Nel caso delle tracce non passanti, esse vengono prolungate fino al lato del poligono o del sotto-poligono su cui giace per poter eseguire il taglio.
A seguito di ogni taglio si ricavano due poligoni distinti dal poligono di partenza. Così si procede fino a che non si è fatta una divisione del poligono per ogni traccia incontrata.

2. STRUTTURE DATI E DOCUMENTAZIONE UML

2.1 STRUTTURE DATI

Le strutture dati scelte per contenere i dati necessari allo sviluppo del problema (vettori, matrici e array), influenzano direttamente le prestazioni computazionali e l'implementazione del programma.

Per cominciare abbiamo costruito un namespace chiamato *'DFNLibrary'* al cui interno abbiamo definito due strutture distinte: *'Fractures'* e *'Traces'*.

La struttura *'Fractures'* è stata progettata per rappresentare l'insieme di fratture all'interno del DFN. Le strutture dati che la compongono sono:

- *'NumberFractures'*: variabile di tipo unsigned int che tiene traccia del numero totale di fratture contenute nel file in lettura
- *'Id'*: variabile di tipo unsigned int che rappresenta l'identificatore univoco per ogni frattura
- *'Vertices'*: vettore di matrici contenente tutti i vertici di ogni poligono. Le matrici hanno 3 righe che rappresentano le coordinate 'x, y, z' dello spazio e un numero dinamico di colonne, poiché il numero dei vertici che definiscono ogni frattura è variabile. Dunque ogni matrice contiene i vertici di una singola frattura
- *'Plane':* vettore di array di 4 elementi di tipo double che rappresentano i coefficienti 'a, b, c, d' dell'equazione del piano in forma parametrica $ax + by + cz + d = 0$ su cui poggia il poligono.

La struttura *'Traces'* è stata progettata per rappresentare l'insieme delle tracce generate dalle intersezioni tra le fratture.
Le strutture dati che la compongono sono:

- *'NumberTraces'*: variabile di tipo unsigned int che tiene traccia del numero totale di tracce che vengono generate dall'intersezione tra poligoni
- *'FracturesId'*: vettore di array di dimensione 2 contenente variabili di tipo unsigned int che contiene gli identificatori delle fratture che generano ogni traccia
- *'Vertices'*: vettore di array di dimensione 2 contenente due vettori tridimensionali che rappresentano le coordinate dei due vertici della traccia. Per conseguenza, ogni array contiene i vertici di ogni traccia
- *'Tips'*: vettore di array contenente due valori booleani: se la traccia risulterà passante si avrà *false*, al contrario le tracce non passanti riporteranno il valore *true*. In particolare si avrà un booleano per ciascuna delle due fratture che genera la traccia.

Inizialmente si era scelto di utilizzare una serie di **mappe** per tutti gli oggetti: gli identificatori erano la chiave e i dati di vertici o i coefficienti erano il valore della mappa. Successivamente si è preferito cambiare le strutture dati e utilizzare dei vettori, in quanto i vettori sono un oggetto meno costoso dal punto di vista computazionale in fase di compilazione quindi l'accesso ai dati risulta più rapido e meno costoso.
Il **vettore** è un oggetto della libreria standard che contiene dati memorizzati in maniera contigua. Si è deciso di utilizzare i vettori quando la dimensione di un oggetto varia durante l'esecuzione del programma, oppure quando è necessario aggiungere elementi in un momento successivo all'allocazione iniziale della memoria, sfruttando il metodo *push_back*.
Sono stati utilizzati i vettori in quanto il costo di accesso al singolo elemento è un *O(1)* rispetto alle liste, che invece essendo memorizzate in maniera non contigua in memoria necessiterebbero dell'utilizzo dei puntatori e il relativo costo di accesso sarebbe un *O(n)*.
Gli **array**, anch'essi oggetti della libreria standard, presentano una dimensione statica, vale a dire che viene definita all'inizio del blocco di appartenenza e non varia nel tempo all'interno del codice, questo perché deve essere nota nel momento in cui avviene la compilazione. Inoltre visto che i dati memorizzati all'interno degli array risiedono in memoria in posizioni contigue l'accesso agli elementi di questa struttura risulta rapido ed efficiente.
Per queste motivazioni si è deciso di memorizzare alcuni oggetti, quali il numero dei coefficienti nell'equazione cartesiana del piano, il numero di fratture che partecipano alla creazione di una traccia e il numero di vertici di un segmento, all'interno di array di dimensione nota.

Successivamente si è costruito il namespace *'PolygonalLibrary'* al cui interno è stata definita la struttura *'PolygonalMesh'*, utilizzata per rappresentare una mesh poligonale composta da celle di dimensione 0 per i vertici, dimensione 1 per i lati e dimensione 2 per i poligoni:
Ogni cella è formata da una variabile *'NumberCellND'* di tipo unsigned int che rappresenta il numero di celle per ciascuna dimensione e da un vettore '*CellNDId'*di interi senza segno che contiene gli identificatori univoci per ciascuna cella di ogni dimensione. Inoltre ogni cella contiene delle strutture dati che la caratterizza e la distingue dalle altre:

- *'Cell0DCoordinates'* è un vettore di vettori di dimensione 3 che rappresentano le coordinate $[x, y, z]$ di ciascun vertice appartenente alla mesh
- *'Cell1DIdVertices'* è un vettore di array di dimensione 2 che contengono gli identificatori dei vertici che sono anche estremi di un segmento
- *'Cell2DVertices'* è un vettore di vettori contenenti gli identificatori dei vertici che formano ciascun poligono
- *'Cell2DEdges'* è un vettore di vettori di dimensione variabile, contenenti gli identificatori dei lati che compongono ciascun poligono.

2.2 DOCUMENTAZIONE UML

Di seguito si riporta l'immagine relativa all'implementazione UML della struttura data al presente progetto.
Si riporta il namespace esterno contenente le strutture, evidenziate dal bordo a linea continua, e l'insieme dei test svolti e delle funzioni scritte, evidenziati dal bordo tratteggiato. Questa distinzione è stata pensata in quanto Test e Funzioni non sono classi o strutture, dunque non appartengono completamente a nessuna delle due. Per conseguenza si è scelto di mantenerle esterne perché attingono e comunicano da entrambe le strutture citate.

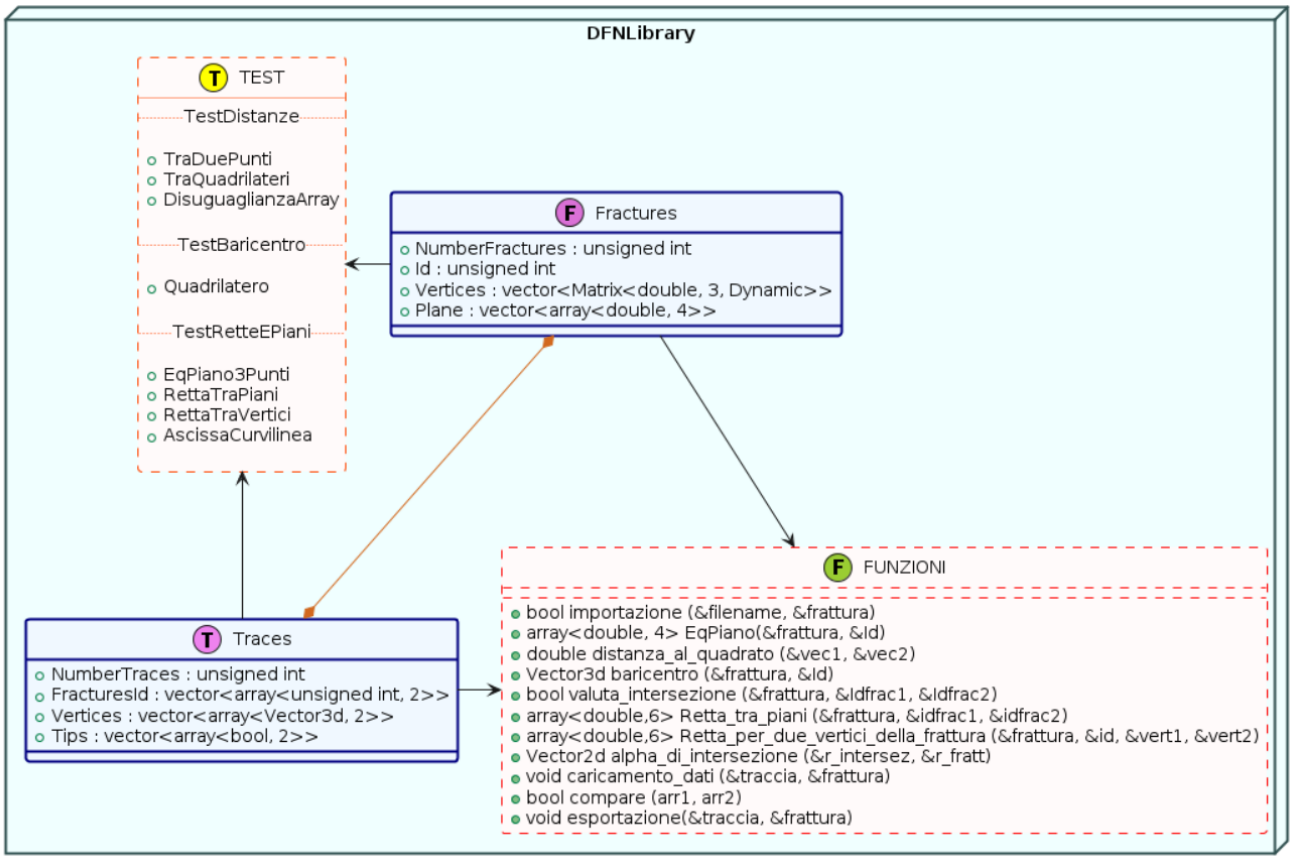

Figura 1: Documentazione UML

3. FUNZIONI E IMPLEMENTAZIONE

All'interno del codice si possono individuare tre sviluppi successivi per giungere all'obiettivo della classificazione delle tracce contenute in ogni frattura: il caricamento dei dati da file, l'individuazione e il calcolo delle tracce e l'esportazione dei risultati ottenuti.
Di seguito si descrivono le funzioni implementate con il relativo scopo e fondamento matematico di ciascuna.

bool importazione (const string& filename, Fractures& frattura)
Questa funzione legge i dati da un file sfruttando la funzione *getline()* per lavorare su una riga alla volta e poter gestire al meglio il salvataggio dei dati all'interno delle strutture dati scelte. Successivamente viene popolata la struct *Fractures*.
Inizialmente si utilizza un oggetto ifstream per aprire il file. Si legge poi il numero di fratture presenti e lo si memorizza nella variabile *numberFractures*. Per ogni frattura la funzione legge l'identificatore della frattura, il numero di vertici che la definiscono e memorizza le coordinate di quest'ultimi in una matrice.
Successivamente vengono calcolati i coefficienti dell'equazione del piano su cui giace ciascun poligono, sfruttando un'altra funzione, e memorizzandoli nell'oggetto *Plane*.
Dopo aver letto ed elaborato tutte le fratture presenti, la funzione chiude il file e restituisce *true* per indicare che l'operazione è stata completata con successo.

array<double, 4> EqPiano (Fractures& frattura, unsigned int& Id)
Grazie a questa funzione si ricavano i coefficienti reali $[a,\ b,\ c,\ d]$ dell'equazione cartesiana del piano in cui giace ogni poligono del sistema considerato.
L'equazione si presenta nella forma $ax + by + cz + d = 0$.
A questo scopo si utilizza la formula dell'equazione del piano passante per tre punti che impiega il calcolo del determinante della matrice A, poi posto uguale a zero. I tre punti considerati sono i primi tre vertici di ogni frattura, da cui il pedice nella matrice seguente.

$$A = \begin{bmatrix} i & j & k \\ x_2 - x_1 & y_2 - y_1 & z_2 - z_1 \\ x_3 - x_1 & y_3 - y_1 & z_3 - z_1 \end{bmatrix}$$

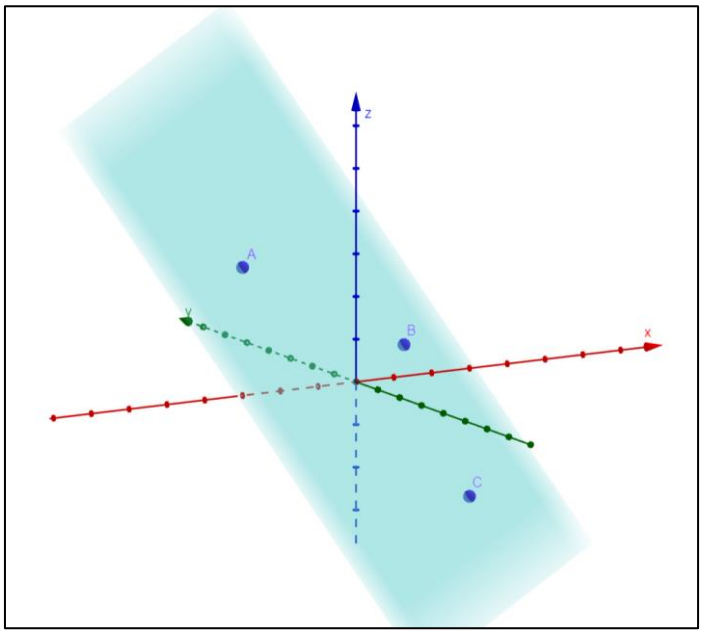

Figura 2: Equazione di un piano passante per tre punti

double distanza_al_quadrato (Vector3d& vec1, Vector3d& vec2)
La funzione restituisce un numero reale che quantifica la distanza tra due punti nello spazio tridimensionale servendosi del Teorema di Pitagora. In questo caso si è scelto di non utilizzare l'elevamento a potenza, per lo

stesso motivo per cui non si è calcolata la radice quadrata per non subire il costo computazionale associato a queste operazioni.
La formula utilizzata è:

$$d^2 = (x_1 - x_2) \cdot (x_1 - x_2) + (y_1 - y_2) \cdot (y_1 - y_2) + (z_1 - z_2) \cdot (z_1 - z_2)$$

Vector3d baricentro (Fractures& frattura, unsigned int& Id1)
Questa funzione serve per calcolare il baricentro delle fratture trattate. Dato *N* il numero di vertici, si lavora separatamente sulle singole coordinate per ciascuna delle tre dimensioni $(x,\ y,\ z)$: ogni vertice contribuisce alla posizione del baricentro *G*. Sommando le singole coordinate di ogni vertice si procede poi dividendo per il numero di vertici, ottenendo così le coordinate del baricentro, poi salvate in un *Vector3d*. In formula matematica si è svolta la seguente operazione:

$$x_G = \sum_{i=0}^{N} \frac{x_i}{N} \qquad y_G = \sum_{i=0}^{N} \frac{y_i}{N} \qquad z_G = \sum_{i=0}^{N} \frac{z_i}{N}$$

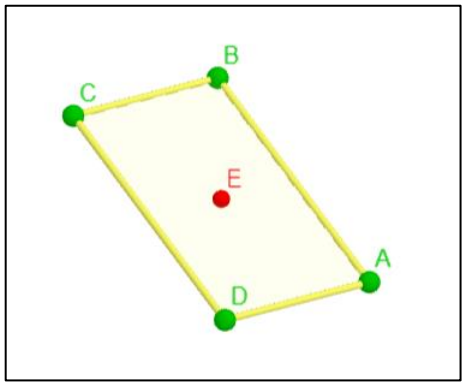

Figura 3:Baricentro di un quadrilatero

bool valuta_intersezione (Fractures& frattura, unsigned int& Idfrac1, unsigned int& Idfrac2)
Questa funzione valuta se due fratture possono intersecarsi, basandosi sulle posizioni reciproche che coppie di poligoni assumono tra loro.
Si comincia individuando il baricentro di ogni poligono e il raggio della sfera circoscritta ad ognuno, considerando come raggio la distanza tra il baricentro e il vertice del poligono da esso più distante.
Prendendo in esame coppie di poligoni, si calcolerà la loro distanza relativa al quadrato testando la distanza al quadrato tra i baricentri dei due poligoni contro il quadrato della somma dei raggi delle sfere circoscritte le due fratture, tenendo conto della tolleranza.
Studiando la distanza relativa si determina se le fratture potrebbero intersecarsi, in tal caso la funzione restituirà *true*, altrimenti le fratture sicuramente non si intersecheranno e la funzione restituirà *false*.
In questo modo si escludono i casi in cui i poligoni sicuramente non si intersecano perché distanti tra loro più della distanza dei loro baricentri. Così procedendo si dovrà studiare un minor numero di possibili intersezioni

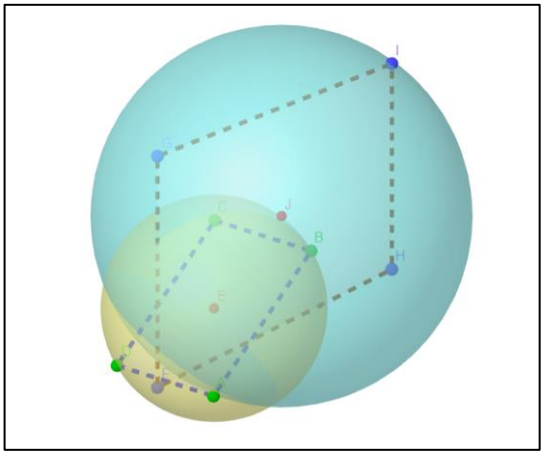
Figura 4: Intersezione tra poligoni

array<double, 6> Retta_tra_piani (Fractures& frattura, unsigned int& idfrac1, unsigned int& idfrac2)
Questa funzione calcola i coefficienti della retta di intersezione tra due piani definiti da fratture distinte e li salva all'interno di un array di dimensione 6, pari al numero di coefficienti nell'equazione della retta. La retta trovata è quella su cui giacciono le tracce.

Si considerino la retta in forma parametrica: $r_{\cap}: \begin{cases} x = a \cdot t + d \\ y = b \cdot t + e \\ z = c \cdot t + f \end{cases}$

e le equazioni di due piani: $P_1: a_1x + b_1y + c_1z + d_1 = 0$ e $P_2: a_2x + b_2y + c_2z + d_2 = 0$

La direzione della retta di intersezione tra i due piani, cioè i coefficienti $[a,\ b,\ c]$, sono determinati dal prodotto vettoriale dei loro vettori normali.

Considerata la matrice $M = \begin{bmatrix} x & y & z \\ a_1 & b_1 & c_1 \\ a_2 & b_2 & c_2 \end{bmatrix}$ si calcolano i coefficienti della retta utilizzando il calcolo

del determinante, da cui $a = (b_1 \cdot c_2 - b_2 \cdot c_1)$, $b = -(a_1 \cdot c_2 - a_2 \cdot c_1)$, $c = (a_1 \cdot b_2 - a_2 \cdot b_1)$

Invece i coefficienti $[d,\ e,\ f]$ sono le coordinate di un punto appartenente alla retta di intersezione.

Tali coordinate sono contenute nel vettore $x = \begin{bmatrix} d \\ e \\ f \end{bmatrix}$ e si ottengono risolvendo un sistema lineare del tipo

$A \cdot x = b$ mediante il metodo di decomposizione LU, dove la matrice

$$A = \begin{bmatrix} a_1 & b_1 & c_1 \\ a_2 & b_2 & c_2 \\ a & b & c \end{bmatrix} \text{ e il vettore } b = \begin{bmatrix} -d_1 \\ -d_2 \\ 0 \end{bmatrix}$$

Il costo computazionale per la risoluzione del sistema lineare risiede prevalentemente nella decomposizione della matrice A ed è pari a $\frac{n^3}{3}$, mentre il costo per la risoluzione del sistema, in confronto, risulta trascurabile.

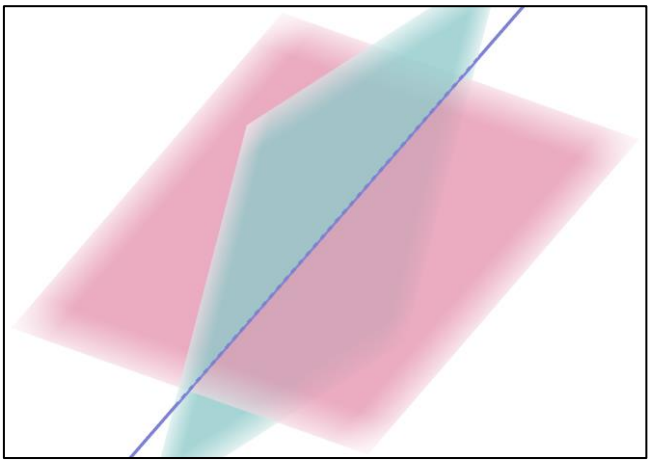

Figura 5: Intersezione tra piani: retta

array<double,6> Retta_per_due_vertici_della_frattura (Fractures& frattura, unsigned int& id, unsigned int& vert1,unsigned int& vert2)
Questa funzione calcola la retta passante per due vertici di una frattura. Si calcola tale retta per poter, in un momento successivo, andare a cercare i vertici delle tracce tramite intersezione con la retta di intersezione tra piani precedentemente individuata.

Dati i vertici $V_1 = \begin{bmatrix} x_1 \\ y_1 \\ z_1 \end{bmatrix}$ e $V_2 = \begin{bmatrix} x_2 \\ y_2 \\ z_2 \end{bmatrix}$ l'equazione della retta cercata è nella forma $r_p: \begin{cases} x = a \cdot t + d \\ y = b \cdot t + e \\ z = c \cdot t + f \end{cases}$

Le componenti sono: $a = (x_2 - x_1)$, $b = (y_2 - y_1)$, $c = (z_2 - z_1)$, mentre i restanti coefficienti sono le coordinate di V_1. Il parametro t è l'ascissa curvilinea della retta.
L'insieme dei coefficienti viene memorizzato in ordine in un array di dimensione 6.

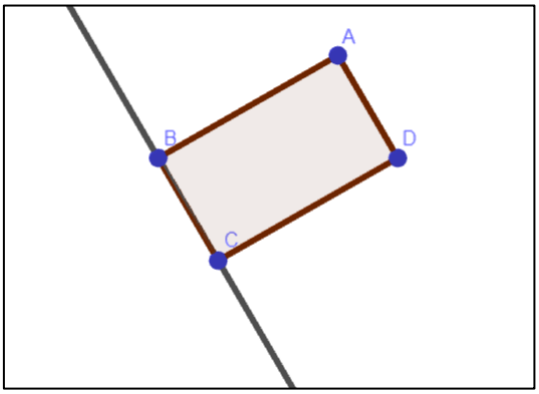

Figura 6: Retta passante per due vertici

Vector2d alpha_di_intersezione (array<double, 6> r_intersez, array<double, 6> r_fratt)

Questa funzione calcola le ascisse curvilinee della retta di intersezione $r_\cap$: $\begin{cases} x = a_\cap \cdot t + d_\cap \\ y = b_\cap \cdot t + e_\cap \\ z = c_\cap \cdot t + f_\cap \end{cases}$
e della retta passante per due vertici di una frattura r_p: $\begin{cases} x = a_p \cdot t + d_p \\ y = b_p \cdot t + e_p \\ z = c_p \cdot t + f_p \end{cases}$ e li salva nel vettore $x = \begin{bmatrix} \alpha \\ \beta \end{bmatrix}$

Questi parametri vengono calcolati risolvendo un sistema lineare del tipo $A \cdot x = b$, dove la matrice

$$A = \begin{bmatrix} a_p & -a_\cap \\ b_p & -b_\cap \\ c_p & -c_\cap \end{bmatrix} \text{ e il vettore } b = \begin{bmatrix} d_\cap - d_p \\ e_\cap - e_p \\ f_\cap - f_p \end{bmatrix}$$

Il sistema lineare viene risolto usando la fattorizzazione QR, perché la matrice in esame è rettangolare con dimensione $m \times n$. Questo metodo ha un costo computazionale di $\frac{2}{3}n^3$.

Vengono calcolate queste due ascisse curvilinee per poter determinare la posizione dei punti di intersezione tra le rette ricavate in precedenza e poter definire quali tra questi punti di intersezione sono vertici di una traccia.

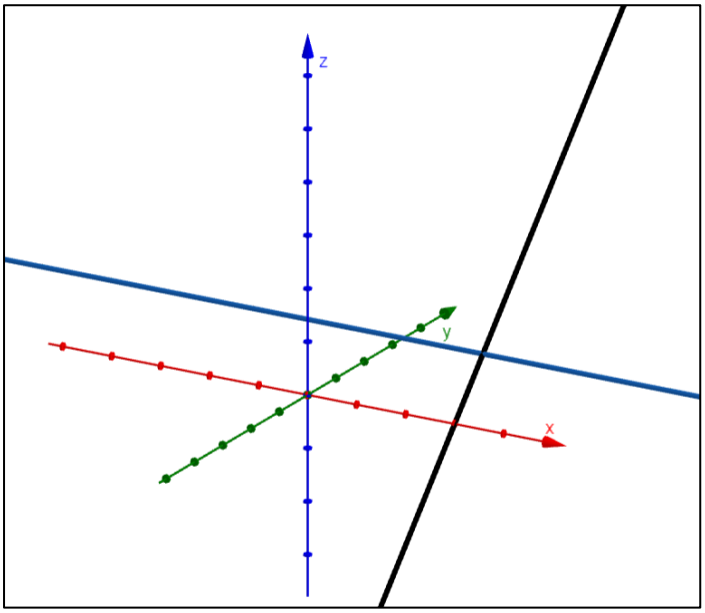

Figura 7: Intersezioni tra rette nel piano

void caricamento_dati(Traces& traccia, Fractures& frattura)
La funzione lavora iterativamente su tutte le fratture: ad ogni iterazione viene considerata una coppia di poligoni diversi. Essa sfrutta tutte le funzioni implementate in precedenza per giungere allo scopo finale del codice: trovare e caratterizzare le tracce generate da una serie di poligoni nello spazio per poi memorizzarle nella struttura appositamente creata *'Traces'*.

Dopo aver verificato la possibilità che vi sia intersezione tra le due fratture considerate, si procede calcolando la retta generata dall'intersezione tra i piani contenenti fratture, escludendo il caso in cui i piani risultino paralleli.
Solo se tale retta esiste, si vanno a calcolare i coefficienti delle rette passanti per coppie di vertici di ciascuna delle due fratture, cioè viene costruita una retta su ogni lato dei poligoni considerati.
Successivamente si valuta l'intersezione tra le rette precedentemente costruite: ogni retta generata dai lati dei poligoni viene testata con la retta di intersezione tra i piani.
Si è utilizzato fino a qui un contatore per inserire all'interno di un array tutte le ascisse curvilinee appartenenti alla retta di intersezione tra piani. In particolare per ogni test tra lato del poligono e retta di intersezione si ricava un'ascissa curvilinea distinta.
Tenendo conto della tolleranza, visto che si sta lavorando su dati di tipo double, si controlla che l'ascissa curvilinea memorizzata appartenga all'intervallo $[0, 1]$: ciò permette di stabilire se il punto di intersezione appartiene o meno al segmento coincidente con il lato del poligono.
Se il contatore assume valore pari a 4, significa che sono state trovate esattamente 4 rette che si intersecano con la retta tra piani, due per ogni poligono coinvolto nell'iterazione. Per questo valore del contatore si procede studiando, in dimensione 1, la posizione reciproca degli intervalli generati dalle ascisse curvilinee dei due poligoni. Se questi si sovrappongono allora si è trovata una traccia generata dai due poligoni considerati.
In tal caso si riconsiderano nuovamente i medesimi poligoni, aggiungendo un contatore per ciascuno, il cui valore equivale al numero di punti di intersezione giacenti sui lati di ogni poligono.
Si sono studiati tre casi possibili, con tre iterazioni distinte e caratterizzate da un diverso valore dei contatori sopra citati:

- Caso 1a: la traccia risulta passante per entrambi i poligoni.
- Caso 1b: la traccia risulta passante per il primo poligono e non passante per il secondo poligono
- Caso 2: la traccia risulta non passante per il primo poligono e passante per il secondo
- Caso 3 la traccia risulta non passante per entrambi i poligoni

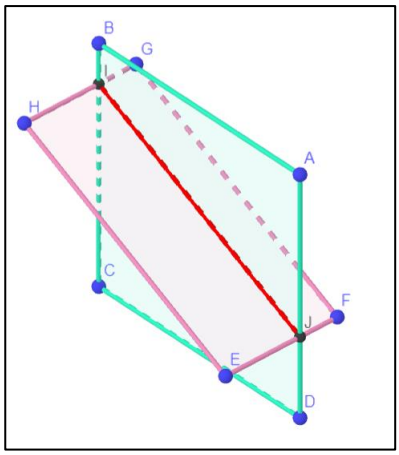

Figura 8: Traccia passante per entrambi i poligoni

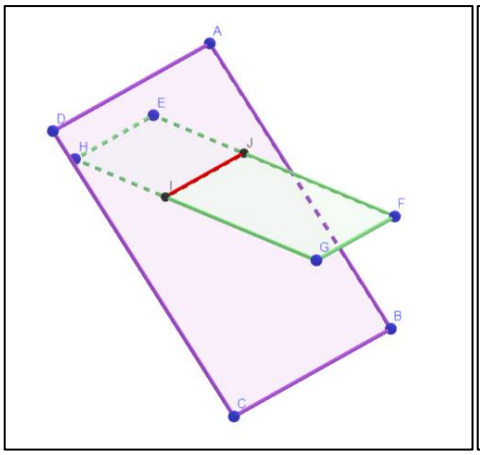

Figura 9: Traccia passante per un poligono e non passante per l'altro poligono

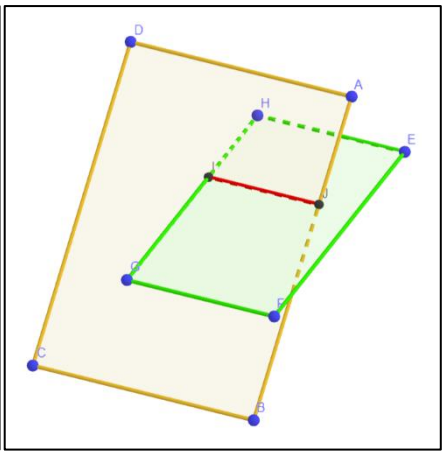

Figura 10: Traccia non passante per entrambi i poligoni

In ciascuno di questi casi, vengono memorizzati il tipo di traccia per ogni poligono, considerando che il valore booleano 0 indica una traccia passante e il valore booleano 1 una traccia non passante, e le informazioni sulla traccia, quali vertici e identificatori dei due poligoni che la generano.

void esportazione (Traces& traccia, Fractures& frattura)
Questa funzione è stata creata al fine di esportare i dati ottenuti in precedenza su un file di testo denominato *'Traces.txt'*.
Per ogni traccia, nel file viene stampato il suo identificatore, gli identificatori delle fratture coinvolte e le coordinate dei suoi due vertici.

Successivamente all'interno del medesimo file viene considerata una frattura alla volta riportando alcune informazioni per ciascuna: il numero complessivo di tracce in essa presenti, il suo identificatore e per ogni traccia vengono elencati identificatore, tipo e lunghezza. Vengono inserite prima le tracce passanti poi quelle non passanti, in ordine di lunghezza decrescente. Per poter ordinare correttamente le tracce si è utilizzata la funzione ***bool compare (array<double, 2> a, array<double, 2> b)*** e il metodo sort. La scelta di quest'ultimo metodo è dovuta al suo costo di ordinamento di un vettore formato da n elementi: $O(n \cdot \log(n))$, il meglio che si può ottenere da un algoritmo il cui scopo è quello di riordinare gli elementi di un vettore.

void caricamento_dati_2(Traces& traccia, Fractures& frattura, PolygonalMesh& mesh)
Questa funzione è stata implementata per la parte 2 del progetto: considerando un poligono alla volta, valuta tutte le tracce precedentemente individuate su di esso e procede al taglio di tale poligono in più sotto-poligoni.
Inizialmente vengono contate, per ogni frattura, le tracce in esso presenti. Tale analisi viene fatta per poter ridurre in partenza il numero di poligoni da analizzare: se sul poligono non è stata individuata alcuna traccia il ciclo termina senza ulteriori processi e passa al poligono successivo.
Per ogni poligono vengono contati il numero totale di punti presenti, siano essi vertici del poligono di origine o vertici dei nuovi sotto-poligoni, e il numero dei sotto-poligoni generati da tutti i tagli eseguiti nel corso del ciclo che opera su tale poligono. Queste quantità vengono inserite, ad ogni iterazione, all'interno della struttura costruita a supporto della mesh poligonale.
Dopo aver salvato i vertici del poligono su cui si sta lavorando, viene introdotto un oggetto di tipo *array<MatrixXd, 2>* che memorizza i due sotto-poligoni generati da un singolo taglio della frattura.
Lavorando su una traccia alla volta, viene studiata la sua posizione nel poligono considerato e in base a questa vengono determinati i due nuovi sotto-poligoni.
Successivamente, per trovare i vertici dei sotto-poligoni, vengono cercati i punti comuni tra la retta formata dall'intersezione dei piani e le rette costruite sui vertici del poligono di partenza. I punti risultanti sono i vertici delle tracce, dunque dei sotto-poligoni.
Infine, usando delle matrici temporanee e di appoggio, si memorizzano e si stampano le coordinate dei vertici dei nuovi poligoni generati.

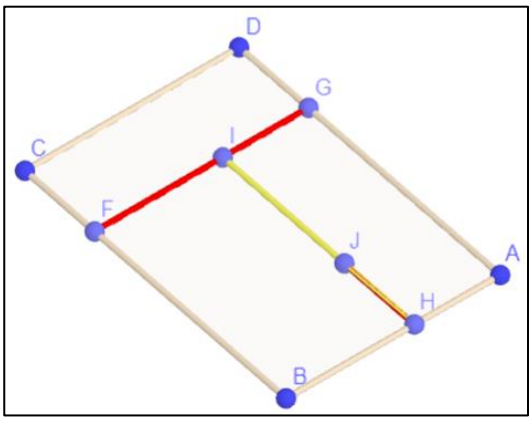

Figura 11: Frattura tagliata da due tracce in 3 sotto-poligoni distinti

4. TESTING

Sfruttando la libreria esterna dei GoogleTest si sono scritti ed eseguiti dei test di unità per verificare la correttezza delle funzioni implementate all'interno del codice sviluppato.
Per costruire i test si è partiti dalle formule matematiche, procedendo con prove pratiche su carta ed eseguendo i passaggi per un caso matematico semplice. Successivamente si è confrontato il risultato dato dalla funzione implementata, con il risultato ricavato manualmente.
Si sono create tre suite: una contenente i test relativi alle funzioni che lavorano sulle distanze, un'altra testa la funzione che calcola il baricentro e la suite più popolata controlla il buon esito delle funzioni inerenti operazioni che lavorano su rette e piani.
Potendo scegliere tra le macro *'EXPECT'* ed *'ASSERT'* si è scelto di utilizzare la seconda opzione in quanto in caso di fallimento, anche di un solo test, si è potuto procedere immediatamente alla correzione dello stesso test fallito.
In base al tipo di dati attesi in output, si sono utilizzate le macro *'ASSERT_EQ'* per tutte le funzioni che restituiscono array o vettori, *'ASSERT DOUBLE EQ'* per la funzione sulla distanza tra due punti, che restituisce un numero reale, infine *'ASSERT TRUE'* per la funzione che valuta la possibilità dell'intersezione tra due fratture e restituisce un valore di tipo booleano e *'ASSERT_FALSE'* per la funzione che confronta il secondo elemento di due array.
Sono state eseguite numerose prove numeriche su carta affiancate da verifiche grafiche, per valutare casistiche diverse. Successivamente solo alcune di esse sono state impiegate nella verifica con i GoogleTest.
Al termine delle varie prove è stato scelto un caso campione presentato nel codice definitivo.

5. CONCLUSIONI

Si riportano alcune osservazioni fatte a seguito dello sviluppo del progetto.
Sono state scelte delle strutture dati che permettessero un'implementazione ottimale del codice, senza tralasciare gli aspetti computazionali di accesso ai dati e la loro allocazione nella memoria in fase di esecuzione.
In fase di sviluppo del codice si è considerata una tolleranza pari a 10^{-10} a fronte del numero di cifre significative dei valori numerici presenti nei file di input e della precisione di macchina pari a 10^{-16}.
Inoltre, in fase di analisi, sono state definite delle funzioni o delle condizioni che permettessero di escludere i seguenti casi: poligoni molto distanti tra loro, fratture giacenti su piani vicini ma paralleli.
Si evidenzia che sono stati utilizzati dei metodi di risoluzione dei sistemi lineari adatti al tipo di matrice con cui si stava lavorando: in caso di matrice quadrata si è scelto il metodo di risoluzione delle eliminazioni di Gauss con decomposizione della matrice $A = LU$, mentre in caso di matrice rettangolare si è scelto il metodo di decomposizione della matrice $A = QR$.
Si è scelto di implementare il progetto suddividendolo in tre parti: importazione dei dati, elaborazione dei dati ed esportazione dei risultati finali.
Infine, in fase di testing, mediante i GoogleTest, si è garantito che le unità logiche, vale a dire le funzioni componenti il codice, fossero verificate e convalidate, assicurando così l'accuratezza dei risultati ottenuti.